Hypergraph Partitioning for Parallel Iterative Solution of General Sparse Linear Systems[*]

Masha Sosonkina[†] Bora Uçar[‡] Yousef Saad[§]

February 1, 2007

Abstract

The efficiency of parallel iterative methods for solving linear systems, arising from real-life applications, depends greatly on matrix characteristics and on the amount of parallel overhead. It is often viewed that a major part of this overhead can be caused by parallel matrix-vector multiplications. However, for difficult large linear systems, the preconditioning operations needed to accelerate convergence are to be performed in parallel and may also incur substantial overhead. To obtain an efficient preconditioning, it is desirable to consider certain matrix numerical properties in the matrix partitioning process.

In general, graph partitioners consider the nonzero structure of a matrix to balance the number of unknowns and to decrease communication volume among parts. The present work builds upon hypergraph partitioning techniques because of their ability to handle non-symmetric and irregular structured matrices and because they correctly minimize communication volume. First, several hyperedge weight schemes are proposed to account for the numerical matrix property called diagonal dominance of rows and columns. Then, an algorithm for the independent partitioning of certain submatrices followed by the matching of the obtained parts is presented in detail along with a proof that it correctly minimizes the total communication volume. For the proposed variants of hypergraph partitioning models, numerical experiments compare the iterations to converge, investigate the diagonal dominance of the obtained parts, and show the values of the partitioning cost functions.

1 Introduction

Although iterative linear system solution techniques require relatively little effort to parallelize, achieving good performance may not be straightforward. Of particular importance for the performance are the matrix-vector product and the quality of the preconditioner. While the structure of the sparse matrix governs the former, the latter is heavily based on the matrix numerical properties and affects the convergence behavior of the iterative method. Conversely, both factors depend on how the sparse matrix is partitioned, which, in general, is dictated by a graph partitioning algorithm employed and by the representation of a matrix in a graph form.

[*]This work was supported in part by NERSC and in part by NSF under grant NSF/ACI-0305120.

[†]Ames Laboratory/DOE, Iowa State University, Ames, IA 50011, (masha@scl.ameslab.gov).

[‡]Department of Mathematics and Computer Science, Emory University, Atlanta, GA 30322 (ubora@mathcs.emory.edu).

[§]Department of Computer Science and Engineering, University of Minnesota, 200 Union Street S.E., Minneapolis, MN 55455, (saad@cs.umn.edu).

The first step in representing the pattern of a matrix is often to symmetrize it, so that an undirected graph is obtained Then, each equation and corresponding unknown is represented by a vertex and each nonzero entry is represented by an edge between the vertices it couples. This "symmetric" graph representation is widely used by a variety of graph partitioners, such as Chaco [21], Distributed Set Expansion (DSE) [34], and MeTiS [23]. By attempting to minimize edge cuts, these algorithms may reduce the length of the boundary part but not necessarily the number of communications, as shown, e.g., in [19]. This could be detrimental to the parallel overhead since establishing an extra communication is typically more expensive than exchanging larger volumes of data at once.

Bipartite graph [19] and hypergraph [8] models have become common tools to partition general sparse matrices. These models can represent non-symmetric matrices and can produce non-symmetric partitions, e.g., a different partition for the rows (equations) and columns (unknowns). Traditionally, the corresponding partitioning techniques target minimizing communication overhead while maintaining load balance during parallel execution of communication-intensive (fine grain) computations such as matrix-vector multiplies, which has been studied in [8, 44].

The effect of the hypergraph partitioning on the convergence of iterative methods, however, is more difficult to capture. This task requires considering not only the sparsity pattern of a matrix, but also its numerical properties. In particular, parallel preconditioning depends greatly on the numerical properties of the partitioned matrix. For example, the simplest parallel preconditioning, Additive Schwarz without overlap incurs no extra communications but may require a certain solution accuracy in the linear sub-system solution phase. On the other hand, in more complex parallel preconditionings, such as incomplete LU (ILU) , the parallel overhead is also due to the communications of matrix data exchange and to possible load imbalances caused by fill-in. Thus, the communication cost and load balancing are still important for parallel preconditioning. It should be noted that, while the communication cost per iteration stays the same, the total communication cost increases with the number of iterations. Therefore, an efficient preconditioner reduces parallel overhead overall by decreasing the number of iterations to convergence.

To the best of out knowledge, only Duff *et al.* [16] use hypergraph partitioning to obtain effective parallel preconditioners. They first reorder the matrix to increase the weight of the diagonal and scale the matrix. Then the following three steps are applied: (1) sparsify the input matrix by dropping nonzeros of magnitude smaller than a tolerance value; (2) apply hypergraph partitioning to the sparsified matrix; (3) construct a preconditioner corresponding to the diagonal blocks resulting from the hypergraph partitioning. These steps are repeated for a range of tolerance values, and at the end, the partition that maximizes the relative Frobenius norm of the preconditioner is used to partition the matrix and hence to build the parallel preconditioner. In a parallel computing environment, the total communication volume during the matrix-vector multiply operations with the coefficient matrix may be large, since the hypergraph partitioning does not have any control on the possible communication due to the dropped nonzeros. Another way to target effective parallel preconditioning might be to add an additional constraint to a hypergraph partitioning algorithm. However, as Pinar and Hendrickson point out in [29], partitioning for some complex objectives cannot, in general, be done by a single partitioning. Most of the time, the complex objective cannot be evaluated before partitioning takes place. They suggest partitioning first for a simple objective and then try to optimize the other(s) in a distinct phase. In the case of partitioning for preconditioiner quality, combining two constraints, structural and preconditioiner balances, may render the partitions sub-optimal, such that neither constraint is properly satisfied. For example, a few extra vertices in a part may spike the matrix-vector multiply operation count in that part. Similarly, an exchange of just a few vertices among parts may improve the uniformity of the preconditioner.

The paper is organized as follows. In Section 2, an overview of commonly used parallel preconditioning techniques is given along with relating matrix reordering to the preconditioner performance. Section 3 describes the proposed hypergraph partitioning techniques to construct parallel preconditioners, while the numerical experiments are given in Section 4.

2 Parallel preconditioning strategies

Consider a linear system

$$Ax = b,$$

where A is a general sparse $n \times n$ matrix, b is the right-hand side vector, and x is the solution vector. An iterative process may be used to obtain x with a certain (given) accuracy. The reduction in the residual $r = b - Ax$ norm is used to monitor the convergence of the iterative method. Preconditioning plays an important role in iterative convergence process. For difficult (ill-conditioned) linear systems, preconditioning may be the only hope in obtaining the solution. Thus, much effort is spent on finding a good preconditioiner given a linear system. In parallel environments, the full spectrum of techniques used in sequential preconditioning may not be practical since many of these techniques are based on incomplete matrix factorization of the entire matrix and are often constructed "on-the-fly" as the factorization progresses. The sheer amount of communication during this process would dominate the parallel overhead. On the other hand, due to the increased processing power, an inexpensive parallel preconditioning, such as Additive Schwarz as shown in Algorithm 2.1 (see, e.g., [31]) , becomes attractive if it leads to a residual norm reduction, albeit in a large number of iterations. This preconditioner requires a solve for the local system which couples the local unknowns only. To improve convergence, Additive Schwarz with overlap [39] may be exploited, such that each subdomain i includes a layer of variables also owned by some neighboring . After solving the local systems, the overlapping variables which are repeated are often averaged in some ways to reach consistency. Restrictive Additive Schwarz (RAS) [6] is a better approach in which, after the local solve, any overlapped variable is simply ignored, so the local subdomain keeps only its own part and discards the rest. This apparently unnatural procedure results in excellent gains in the number of iterations needed to converge, see [6] for details.

ALGORITHM **2.1.** *Additive Schwarz in subdomain* i

```
1.  Update local residual $r_i = (b - Ax)_i$.
2.  Solve $A_i \delta_i = r_i$.
3.  Update local solution $x_i = x_i + \delta_i$.
```

When the number of iterations to convergence becomes large with Additive Schwarz, more sophisticated preconditioning techniques may be considered. In particular, distributed Schur Complement (dSC) techniques [33] often lead to superior convergence properties while retaining good parallelism. Briefly, the dSC techniques are more effective than their standard Additive Schwarz counterparts because, in contrast with Additive Schwarz, they attempt to solve a smaller global system which only couples local and remote unknowns from neighboring subdomains. This solve is preceded by a communication phase, which is akin to the one used in the matrix-vector multiply to exchange values for the boundary unknowns. Thus, dSC techniques incur larger parallel overhead compared with the related Additive Schwarz procedure from which it is derived. The details of the dSC construction may be found in [33] and their implementation in the vertex-based row-wise partitioning in [35].

Distributed Schur complement and Additive Schwarz preconditioners are both based on the incomplete LU factorization of the matrix A_i which is local to subdomain $i,\ i = 1, \ldots, K$. For ill-conditioned linear systems, the incomplete factorizations may be inaccurate in the sense that $\|A_i - L_iU_i\|$ is not small, or unstable, in the sense that $\|(L_iU_i)^{-1}\|$ is huge. In [11], a strong correlation between instability of the preconditioner and the size of $\mathcal{E} = \log(\|(LU)^{-1}\|_{\text{inf}})$ is shown, and this is suggested as a practical means of gauging the quality of a preconditioner. This rough measure of instability can be inexpensively computed as $\mathcal{E} = \log(\|(LU)^{-1}e\|_1)$, where e is a vector of all ones and LU is a product of incomplete LU factors of A. For distributed preconditioners, a *per subdomain* value $\mathcal{E}_i$ is considered as $\mathcal{E}_i = \log(\|(L_iU_i)^{-1}e_i\|_1)$, where L_i and U_i are the incomplete factors local to subdomain i [40].

2.1 Nonsymmetric permutations: diagonal dominance PQ orderings

In recent years, a number of new reordering strategies have been designed to improve the robustness of preconditioning techniques. A major distinction between these strategies and classical ones is that they sacrifice symmetry in the reorderings, i.e., they reorder rows and columns differently. For problems arising from partial differential equations, this is usually considered a poor idea. However, in cases of extreme indefiniteness, these techniques can be of great help. Recent papers [14, 15, 32] indicate that these methods can lead to significant improvements in the preconditioner robustness. The code MC64 [14, 15] performs a one-sided permutation of the columns (or rows) of A in a preprocessing stage. The leading criterion used here is that the column permutation q should be such that

$$\prod_{i=1}^{n} |a_{i,q(i)}| \text{ is maximized.}$$

This follows work by Olschowska and Neumaier [28] who initially developed this strategy as a means of avoiding pivoting in Gaussian elimination. These authors translate the above optimization problem into

$$\min_q \sum_{i=1}^{n} c_{i,q(i)} \quad \text{with } c_{ij} = \begin{cases} \log\left[\frac{\|a_{:,j}\|_\infty}{|a_{ij}|}\right] & \text{if } a_{ij} \neq 0 \\ +\infty & \text{else.} \end{cases}$$

Once this problem is solved resulting in a permutation, an ILU factorization is performed on the permuted matrix usually yielding a more stable preconditioner.

In [32] a dynamic procedure was advocated instead of this static approach. The technique referred to as diagonal dominance PQ orderings (ddPQ orderings) is based on a simple strategy which uses the ARMS preconditioner framework [36] to extract 2-sided permutations (P for rows and Q for columns). In the following we summarize the procedure. Details can be found in [32]. The main idea is to generalize the ARMS ordering by using 2-sided permutations. So, the matrix A is permuted (on both sides) and the resulting matrix has following block structure:

$$PAQ^T = \begin{pmatrix} B & F \\ E & C \end{pmatrix}. \tag{1}$$

No particular structure is assumed for the B block. Instead, the pair of permutations P, Q is selected so that the B block has the "most diagonally dominant" rows and few nonzero elements (to reduce fill-in). This principle can now be carried to a multilevel framework of ARMS. As in ARMS, at the l-th level we reorder matrix as shown above and then we carry out the block factorization "approximately"

$$P_l A_l Q_l^T = \begin{pmatrix} B_l & F_l \\ E_l & C_l \end{pmatrix} \approx \begin{pmatrix} L_l & 0 \\ E_l U_l^{-1} & I \end{pmatrix} \times \begin{pmatrix} U_l & L_l^{-1} F_l \\ 0 & A_{l+1} \end{pmatrix}, \qquad (2)$$

where

$$\begin{aligned} B_l &\approx L_l U_l \\ A_{l+1} &\approx C_l - (E_l U_l^{-1})(L_l^{-1} F_l) \ . \end{aligned}$$

Note that the motivation for this strategy is that it is critical to have an accurate and well-conditioned B block, see [4, 5]. The case when B is of dimension 1 corresponds to an inexpensive form of a complete pivoting ILU. The procedure can therefore be viewed from the angle of a incomplete LU factorization with complete pivoting.

Let p_i and q_i, $1 \le p_i, q_i \le n$ be matrix row and column numbers, respectively. Define a matching set $\mathcal{M} = \{(p_i, q_i) \mid p_i \ne p_j, \text{ and } q_i \ne q_j, \ \forall j \ne i, \ i, j = 1, \ldots, n_{\mathcal{M}} \text{ and } n_{\mathcal{M}} \le n\}$. In the case when the number of pairs $n_{\mathcal{M}} = n$, we have a (full) permutation pair (P, Q). A *partial* matching set can be easily completed into a full pair (P, Q) by a greedy approach. Algorithm 2.2 presents the three stages to find the PQ ordering.

ALGORITHM **2.2.** *PQ ordering construction*

```
1.  Preselection: Filter out poor rows (in the sense of diagonal dominance).
                  The nodes are sorted—in descending order—
                      according to an inexpensive priority rule
                      for consideration in the next phase.
2.  Matching: Scan the candidate entries in order given by preselection;
                  Accept them into set $\mathcal{M}$, or reject them.
3.  Completion: Complete the matching set into a complete pair $(P, Q)$
                      by a greedy algorithm.
```

Once the PQ ordering is constructed, the matrix in (1) is re-ordered, and the C block corresponds to the rejected nodes. Next, the factorization (2) is performed and the process is repeated recursively on the Schur complement. There are many possible variants for all three stages in Algorithm 2.2.

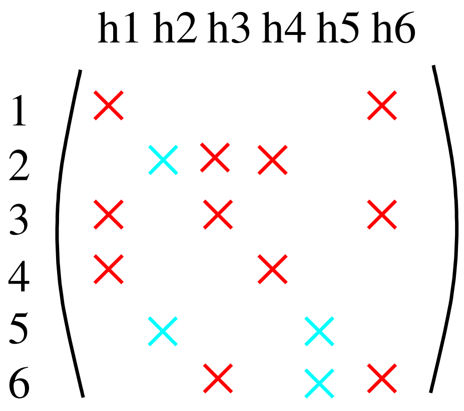
We now mention a few of the choices used for the diagonal dominance criterion. These are used in the preselection as well as in the matching phases. For example, in the preselection, one can decide to sort entries a_{ij} according to the scalars:

$$\rho_{ij} = \frac{|a_{ij}|}{\|a_{i,:}\|_1},$$

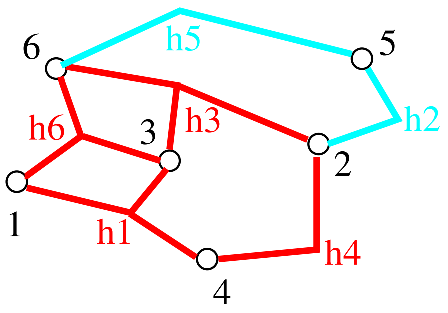
where $\|a_{i,:}\|_1$ is the 1-norm of the row i, following the Matlab notation. If a_{ij} is permuted into a diagonal entry, then ρ_{ij} would represent a diagonal dominance strength of this entry. In particular, if $\rho_{ij} > 1/2$, then the row would become strictly diagonally dominant. Sorting all the entries by diagonal dominance has the major weakness that it does not take into consideration potential fill-in. For this reason, the criterion is often altered by multiplying the denominator by the number of nonzero entries in the row:

$$\rho_{ij} = \frac{|a_{ij}|}{\|a_{i,:}\|_1 \ \times \mathrm{nnz}(a_{i,:})}.$$

Finally, it is clear that column diagonal dominance can be used instead of row diagonal dominance. One can also combine the two.



(a)



(b)

Figure 1: Column-net hypergraph representation

3 Hypergraph partitioning models

To represent a sparse $n \times n$ matrix A, we use a column-net hypergraph model proposed in [8]. They define hypergraph $G = (V, H)$, where $V = \{v_1, \ldots, v_n\}$ is the set of vertices representing the matrix rows and $H = \{h_1, \ldots, h_n\}$ is the set of hyperedges representing the columns, such that h_j contains the vertices $\{v_i \mid a_{ij} \in A \text{ and } i = 1, \ldots, n\}$. Figure 1 shows a sample matrix and the associated column-net hypergraph model, in which the vertices are shown with circles and hyperedges as lines connecting to the vertices. In the column-net hypergraph model, weights can be associated with the vertices and hyperedges.

$\Pi = \{V_1, V_2, \ldots, V_K\}$ is a K-way partition of the vertex set if

$$V = \cup_{i=1}^{K} V_i \quad \text{and} \quad V_i \cap V_j = \emptyset \quad \forall i, j, \text{ such that } i \neq j.$$

In Π, the connectivity set Λ_j of an hyperedge h_j is the set of parts in which h_j has vertices. The connectivity λ_j of a hyperedge h_j is the cardinality of Λ_j. A hyperedge h_j is said to be cut if $\lambda_j > 1$. For example, if in Figure 1(b) $V_1 = \{1, 3, 4\}$ and $V_2 = \{2, 5, 6\}$ then the hyperedge h_6 has connectivity two and hence it is a cut hyperedge, while hyperedge h_1 has connectivity one.

In hypergraph partitioning problem, the objective is to minimize the cutsize C_Π which may be defined in several ways using the weight c_j of hyperedge h_j:

$$C(\Pi) \equiv C^{\lambda}(\Pi) = \sum_{j \in H} c_j(\lambda_j - 1) \quad \text{or} \quad C(\Pi) \equiv C^{c}(\Pi) = \sum_{j \in H, \lambda_j > 1} c_j. \tag{3}$$

These two objective functions are widely used in the VLSI community [26] and also in the scientific computing community (see for example [2, 8, 41, 44, 3]). The partitioning constraint is to satisfy a balancing constraint on part weights. Typically the weight of a part is the sum of the weights of vertices in that part in partition Π. A variant of this problem is the multi-constraint hypergraph partitioning [10, 25] in which each vertex has a vector of weights associated with it. The objective is the same as in (3), and the partitioning constraint is to satisfy a balancing constraint associated with each weight.

A distinct advantage of the hypergraph models over the standard and bipartite graph models is that the partitioning objective of minimizing $C^{\lambda}(\Pi)$ is an exact measure of the total communication volume, whereas the objective in the graph models is an approximation [8, 18, 19, 20]. The hypergraph models are said to be more expressive than the traditional graph models as they can produce non-symmetric partitions [8, 18, 19, 20] and can represent more involved computational dependencies [42, 43], e.g., successive matrix-vector multiplies.

There are two common approaches to define the vertex weights. The first one sets the weight of the vertex v_i to be equal to the number of nonzeros in the ith row of the matrix. This approach aims to achieve load balance among processors for the matrix-vector multiply operations. The second one sets a unit weight for each vertex. This approach aims to achieve the balance on the number of unknowns per processor. For preconditioning, neither of these two approaches necessarily lead to balanced workload, mainly because the amount of computations in preconditioning depends much on the numerical values of the matrix nonzeros.

In the rest of this section, we investigate three hypergraph based matrix partitioning methods. They target better parallel preconditioners and load balancing of preconditioner operations. As is customary in scientific computing community, we are interested in K-way partitioning of the matrices, where K is the number of processors, such that the ith part is assigned to the ith processors. In Section 3.1, we present schemes with weights on hyperedges in an attempt to favor a certain type of hyperedges during partitioning. In Section 3.2, we investigate two different partitioning techniques which try to balance the difficulty of local subsystems.

3.1 Assigning hyperedge weights

Weights on hyperedges and/or on vertices may play an important role in using hypergraph partitioning to construct a good preconditioning. In particular, incorporating specific numerical properties, such as diagonal dominance, into hypergraph models may prove beneficial. Consider a weighted version $G_w = (V, H, W_v, W_h)$ of the hypergraph G defined earlier, in which W_v is the set of vertex weights and W_h is the set of hyperedge weights.

The weight w_j of the hyperedge h_j is based on the notion of *weak diagonal dominance* in a relative sense. Define

$$\tau_j = \frac{a_{jj}}{\|a_{:j}\|_1} \quad \text{and} \quad \tau_j' = \frac{\tau_j}{\max_i |\tau_i|}.$$

Column j (the hyperedge h_j) is deemed sufficiently diagonally dominant (denoted as $\boldsymbol{\mathcal{D}}$) if $\tau_j' \geq \Delta$, where $0 < \Delta \leq 1.0$. For example, in Figure 1, the $\boldsymbol{\mathcal{D}}$ hyperedges might be h_1, h_3, h_4, and h_6 (light-colored), whereas $\tilde{\boldsymbol{\mathcal{D}}}$ might be h_2 and h_5 (dark-colored) based on some Δ.

We have explored different weight schemes in which either $\boldsymbol{\mathcal{D}}$ or $\tilde{\boldsymbol{\mathcal{D}}}$ hyperedges are heavily weighted to tune the partitioning algorithms to favor $\boldsymbol{\mathcal{D}}$ or $\tilde{\boldsymbol{\mathcal{D}}}$ hyperedges, respectively. In the former case, when the $\boldsymbol{\mathcal{D}}$ hyperedges are to be kept in the same partition, a local submatrix will exhibit better weak diagonal-dominance. This is beneficial for solving local systems when constructing distributed preconditioning, such as Additive Schwarz. In the latter case, when the partitioner attempts to keep the $\tilde{\boldsymbol{\mathcal{D}}}$ hyperedges in the same partition, the interfaces may contain diagonal-dominant columns, i.e., columns with small off-diagonal matrix entries. Thus, when these entries are disregarded or not fully considered in preconditioning, the information and accuracy loss may be tolerable.

Two weight schemes have been tried—one emphasizing the balance between the cardinality of the hyperedge and its magnitude and the other considering directly the relative weak diagonal dominance τ' of the hyperedge. They are denoted W_h^s and W_h^τ, respectively, and defined as follows.

Definition 3.1. *A hyperedge h_j is assigned a weight $w_j^s \in W_h^s$, where*

$$w_j^s = \begin{cases} 1,000 \sum_{v_i \in h_j} |v_i|/|h_j| & \text{if } h_j \text{ is } \boldsymbol{\mathcal{D}}\ (\tilde{\boldsymbol{\mathcal{D}}}); \\ 1 & \text{otherwise}. \end{cases}$$

Definition 3.2. *A hyperedge h_j is assigned a weight $w_j^\tau \in W_h^\tau$, where*

$$w_j^\tau = \begin{cases} 1,000\tau' & \text{if } h_j \text{ is } \boldsymbol{\mathcal{D}}\ (\tilde{\boldsymbol{\mathcal{D}}}); \\ 1 & \text{otherwise}. \end{cases}$$

3.2 Separate partitioning techniques

It may be observed that, in general, irregular-structured matrices do not exhibit good diagonal-dominance of rows or columns. Thus, both $\boldsymbol{\mathcal{D}}$ and $\tilde{\boldsymbol{\mathcal{D}}}$ hyperedges may be present in a matrix hypergraph, although the precise ratio of the $\boldsymbol{\mathcal{D}}$ and $\tilde{\boldsymbol{\mathcal{D}}}$ hyperedges depends on the user-supplied value of Δ when relative weak diagonal dominance τ' is considered. For the weight schemes W_h^s and W_h^τ, it may happen that some processors contain significantly more hyperedges of a certain type than others do so. Thus, some local system solvers will work harder than others, lagging behind in time and/or accuracy and creating preconditioner workload imbalance. We propose distributing the $\boldsymbol{\mathcal{D}}$ and $\tilde{\boldsymbol{\mathcal{D}}}$ hyperedges among processors evenly in an attempt to avoid the varying difficulty among local systems. We consider two approaches. One uses a multi-constraint hypergraph partitioning tool as black-box, the other decomposes the problem into two sub-problems and then combines the solutions. Both approaches assume a 2×2 block structure on the coefficient matrix A

$$A_{BL} = \begin{bmatrix} A_{dd} & A_{d\sigma} \\ A_{\sigma d} & A_{\sigma\sigma} \end{bmatrix}. \qquad (4)$$

The block structure is obtained by symmetrically permuting the matrix A, such that the $\boldsymbol{\mathcal{D}}$ columns precede the $\tilde{\boldsymbol{\mathcal{D}}}$ ones (due to symmetrical permutation the rows corresponding to the $\boldsymbol{\mathcal{D}}$ columns precede the rows corresponding to the $\tilde{\boldsymbol{\mathcal{D}}}$ columns). In other words, the submatrix A_{dd} contains all nonzeros a_{ij} where columns i and j are $\boldsymbol{\mathcal{D}}$. Likewise, $A_{\sigma\sigma}$ contains all nonzeros a_{ij} where columns i and j are $\tilde{\boldsymbol{\mathcal{D}}}$. The submatrices $A_{d\sigma}$ and $A_{\sigma d}$ contain nonzeros a_{ij} where either the column i or j is $\boldsymbol{\mathcal{D}}$.

3.2.1 A multi-constraint formulation

In this approach, we use the column-net hypergraph model [8] with a two-constraint formulation. For each vertex v_i, the weight vector $\langle 1, 0 \rangle$ is used if the corresponding row i is in the first row block of A_{BL} (4), otherwise the weight vector $\langle 0, 1 \rangle$ is used. Partitioning the hypergraph into K parts will minimize the total communication volume in the matrix-vector multiplies with the coefficient matrix A. Maintaining balance on the part weights in terms of the both weights of the vertices will achieve balance on the number of unknowns per processor and also the number of rows corresponding to $\boldsymbol{\mathcal{D}}$ and $\tilde{\boldsymbol{\mathcal{D}}}$ columns per processor. Furthermore, if the rows have almost equal number of nonzeros, then this approach will likely achieve computational load balance among the processors for the matrix-vector multiply operations. Note that the weight scheme proposed in Section 3.1 can also be used.

3.2.2 Independent partitioning followed by matching

The state-of-the-art methods for standard graph and hypergraph partitioning problem do not perform equally well under the multi-constraint formulation. For example, in recursive bisection based approaches, the cutsize of a two-constraint partitioning is 20% to 30% worse than that of the standard partitioning of the same graph [24, 45] and hypergraph [1, 43]. A direct K-way hypergraph partitioning method is proposed in [1] which performs considerably better than recursive bisection based approaches in multi-constraint formulation. However, the cutsize in two-constraint partitioning still lags behind that of the standard partitioning by about 10%.

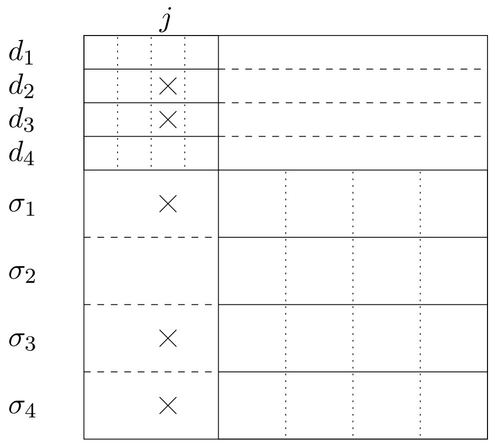
In the proposed method, we partition the rows corresponding to $\boldsymbol{\mathcal{D}}$ and $\tilde{\boldsymbol{\mathcal{D}}}$ columns independently. For this purpose, we use the column net hypergraph models G_{dd} and $G_{\sigma\sigma}$ of the submatrices A_{dd} and $A_{\sigma\sigma}$ (see (4)), respectively. We partition these two hypergraphs into K parts as $\Pi_d = \{d_1, \ldots, d_K\}$ and $\Pi_\sigma = \{\sigma_1, \ldots, \sigma_K\}$, respectively, with the objective of minimizing the $C^\lambda(\Pi)$. If the matrix A_{BL} has zeros in the main diagonal, we modify the hyperedges in G_{dd} and $G_{\sigma\sigma}$ to contain the corresponding row vertices. This is a common practice [8, 42, 44] which guarantees that the cutsizes $C^\lambda(\Pi_d)$ and $C^\lambda(\Pi_\sigma)$ correspond exactly to the volume of communication regarding A_{dd} and $A_{\sigma\sigma}$. Each processor will be assigned a part from each partition, e.g., a set of rows corresponding to $\boldsymbol{\mathcal{D}}$ columns and a set of rows corresponding to $\tilde{\boldsymbol{\mathcal{D}}}$ columns. Note that since the two K-way partitions maintain balance on part weights, this approach will achieve (per processor) balance on the number unknowns and also on the number of $\boldsymbol{\mathcal{D}}$ and $\tilde{\boldsymbol{\mathcal{D}}}$ columns, after the parts are matched.

In matching a pair of row parts, one from Π_d and the other from Π_σ, the total volume of communication in matrix-vector multiplications needs to be minimized. Specifically, the proposed method tries to minimize the total volume of communication using a divide-and-conquer approach. Firstly, it addresses the volume of communication within the submatrices A_{dd} and $A_{\sigma\sigma}$, secondly, combines the solutions to address the total volume of communication. A matching problem of the same kind arises in a different context [1], where the problem is formulated as a maximum weight bipartite matching problem. Here, we provide yet another formulation based on the minimum weight perfect matching in bipartite graphs—also known as the assignment problem.

Algorithm 3.1 outlines the process, denoted IPM, of the independent partitioning followed by matching. We define an edge-weighted complete bipartite graph $G_B = (\Pi_d, \Pi_\sigma, E, \Omega)$ (line 5 of Algorithm 3.1), where $\Pi_d = \{d_1, \ldots, d_K\}$ and $\Pi_\sigma = \{\sigma_1, \ldots, \sigma_K\}$ represent the sets of parts in the corresponding vertex partitions Π_d and Π_σ; $E = \Pi_d \times \Pi_\sigma$ is the set of all possible edges. Ω is the set of edge weights resulted from the AssignEdgeWeights procedure (line 4) as follows.

ALGORITHM **3.1.** *IPM: Independent Partitioning and Matching*

```
1.  $G_{dd} \leftarrow$ column-net hypergraph of $A_{dd}$
    $G_{\sigma\sigma} \leftarrow$ column-net hypergraph of $A_{\sigma\sigma}$
2.  $\Pi_d = \{d_1, \ldots, d_K\} \leftarrow \mathrm{partition}(G_{dd}, K)$
    $\Pi_\sigma = \{\sigma_1, \ldots, \sigma_K\} \leftarrow \mathrm{partition}(G_{\sigma\sigma}, K)$
3.  Compute $\Lambda_d(j)$ and $\Lambda_\sigma(j)$ for all columns $j$
4.  $\Omega \leftarrow \mathrm{AssignEdgeWeights}(\Pi_d, \Lambda_d, \Pi_\sigma, \Lambda_\sigma, d, \sigma, A)$
5.  Define $G_B = (\Pi_d, \Pi_\sigma, E, \Omega)$.
6.  $M \leftarrow$ a minimum weight perfect matching in $G_B$
7.  for each $\langle d_k, \sigma_\ell \rangle \in M(G_B)$ assign the corresponding
    rows to processor $P_k$.
```



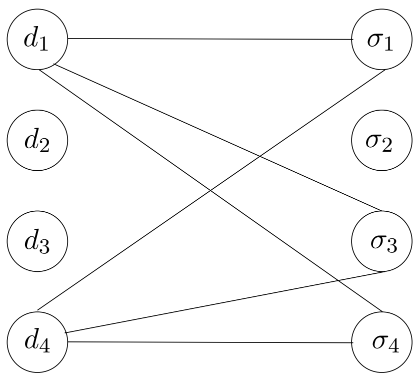


Figure 2: (a) Four-way row-wise partitions (solid lines) $\Pi_d = \{d_1, d_2, d_3, d_4\}$ and $\Pi_\sigma = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$ on A_{dd} and $A_{\sigma\sigma}$. The row-wise partitions are shown with horizontal solid lines. These partitions are extended into the blocks $A_{d\sigma}$ and $A_{\sigma d}$ and shown with horizontal dotted lines. Column partitions conforming to the row-wise partitions are shown with vertical dotted lines. The column j of type $\mathcal{D}$ has connectivities $\Lambda_d(j) = \{d_2, d_3\}$ and $\Lambda_\sigma(j) = \{\sigma_1, \sigma_3, \sigma_4\}$. (b) The bipartite graph $G_B = (\Pi_d, \Pi_\sigma, E, \Omega)$. For the column j, the displayed edges lines bear a unit weight.

Initially, all the edge weights, $\omega(k, \ell) \in \Omega; \text{for all } k \in \Pi_d \text{ and } \ell \in \Pi_\sigma$ are zero. Then AssignEdgeWeights processes the columns of type $\boldsymbol{\mathcal{D}}$ followed by the columns of type $\tilde{\boldsymbol{\mathcal{D}}}$ using UpdateEdgeWeights Algorithm 3.2 in each case. Consider, for example, the $\boldsymbol{\mathcal{D}}$ columns. For each column $j \in \boldsymbol{\mathcal{D}}$ having nonzeros in $A_{\sigma d}$, let $\pi_d(j)$ denote the part in partition Π_d to which the corresponding row i belongs. Let $\Lambda_d(j)$ and $\Lambda_\sigma(j)$ denote, respectively, the sets of parts connected by hyperedge h_j (column j) according to the partitions Π_d and Π_σ. Note that $\pi_d(j) \in \Lambda_d(j)$ due to the condition $v_j \in h_j$ (i.e., zero-free diagonal assumption). Note also that the column j does not appear as a hyperedge in the hypergraph $G_{\sigma\sigma}$, and hence the connectivity set $\Lambda_\sigma(j)$ of hyperedge h_j according to the partition Π_σ, is computed with respect to the row-wise partition on $A_{\sigma d}$ induced by the partition Π_σ. For each part pair (d_k, σ_ℓ), where $d_k \in \Pi_d \setminus \Lambda_d(j)$ and $\sigma_\ell \in \Lambda_\sigma(j)$, we add one to the weight of the edge $(d_k, \sigma_\ell) \in E$. The rationale is that a part *not* connected by hyperedge h_j in Π_d necessitates an additional unit of communication if matched to a part connected by hyperedge h_j under the partition Π_σ, i.e., a part having nonzeros in column j in the submatrix $A_{d\sigma}$. Figure 2 illustrates these points on a partially shown hypothetical matrix. Consider the column j in Figure 2(a). Assume that its corresponding row j is in d_3, e.g., $\pi_d(j) = d_3$, and that j has connectivity sets $\Lambda_d(j) = \{d_2, d_3\}$ and $\Lambda_\sigma(j) = \{\sigma_1, \sigma_3, \sigma_4\}$. Since $\Pi_d \setminus \Lambda_d(j) = \{d_1, d_4\}$, it contributes one to the weights of the displayed edges (Figure 2(b)), e.g., to the edges belonging to the set $\{d_1, d_4\} \times \{\sigma_1, \sigma_3, \sigma_4\} \in E$.

ALGORITHM **3.2.** *UpdateEdgeWeights*$(\Omega, \Pi_d, \Lambda_d, \Pi_\sigma, \Lambda_\sigma, T, O, A)$

```
1.  for each column $j$ of type T, $j \cap A_{OT} \neq 0$
2.      if $T = d$ then   $S_d \leftarrow \Pi_d \setminus \Lambda_d(j)$ ; $S_\sigma \leftarrow \Lambda_\sigma(j)$
3.      else              $S_d \leftarrow \Lambda_d(j)$ ; $S_\sigma \leftarrow \Pi_\sigma \setminus \Lambda_\sigma(j)$
4.      for each $(d_k, \sigma_\ell) \in S_d \times S_\sigma$
5.          $\omega(d_k, \sigma_\ell) \leftarrow \omega(d_k, \sigma_\ell) + 1$
```

Recall that $C^\lambda(\Pi_d)$ and $C^\lambda(\Pi_\sigma)$ denote, respectively, the cutsize of the partitions of the hypergraphs G_{dd} and $G_{\sigma\sigma}$. Let $C_B(M)$ denote the cost of a minimum weight perfect matching

M in the bipartite graph G_B described above. Then we have the following theorem establishing the relation among the total communication volume and these three quantities.

Theorem 3.1. *Total volume of communication in matrix-vector multiplies with A is equal to $C^{\lambda}(\Pi_d) + C^{\lambda}(\Pi_\sigma) + C_B(M)$.*

Proof: Consider a column whose nonzeros are confined within A_{dd} or $A_{\sigma\sigma}$. Then the communication volume regarding the associated vector entry is encapsulated in the cutsize of the corresponding hypergraph partitioning, and no edge in G_B is assigned a weight due to that column. For the columns having nonzeros in $A_{d\sigma}$ or $A_{\sigma d}$ (each column has at least one nonzero in either A_{dd} or $A_{\sigma\sigma}$ due to zero-free diagonal assumption), we discuss only the $\boldsymbol{\mathcal{D}}$ columns—$\tilde{\boldsymbol{\mathcal{D}}}$ columns can be treated similarly.

Consider a column j of type $\boldsymbol{\mathcal{D}}$ having nonzeros in A_{dd} and $A_{\sigma d}$. We need to show that messages that are sent by the processor $P_{\pi_d(j)}$ regarding the vector entry x_j are accounted for. Consider a pair of parts $\langle d_k, \sigma_\ell \rangle$ matched by the bipartite matching algorithm where $d_k \neq \pi_d(j)$. The processor $P_{\pi_d(j)}$ sends a message to the processor P_k corresponding to $\langle d_k, \sigma_\ell \rangle$ iff $d_k \in \Lambda_d(j)$ or $\sigma_\ell \in \Lambda_\sigma(j)$. If $d_k \in \Lambda_d(j)$, then by the definition of the edge weights, the edge $(d_k, \sigma_\ell) \in E$ does not bear a weight corresponding to j, regardless of σ_ℓ being in $\Lambda_\sigma(j)$ or not because the cost of sending x_j from $P_{\pi_d(j)}$ to P_k is included in $C^{\lambda}(\Pi_{dd})$. If $d_k \notin \Lambda_d(j)$, but $\sigma_\ell \in \Lambda_\sigma(j)$, then the communication volume of sending x_j from $P_{\pi_d(j)}$ to P_k is *not* included in $C^{\lambda}(\Pi_{dd})$. By the definition of the edge weights, the edge $(d_k, \sigma_\ell) \in E$ bears a unit weight for the column j. Therefore, the cost of sending x_j from $P_{\pi_d(j)}$ to the processor P_k is included in the weight of the matching in the bipartite graph. Q.E.D.

To see how the theorem works, consider again Figure 2 and assume that $M = \{\langle d_i, \sigma_i \rangle : i = 1, \ldots, 4\}$ is a minimum weight perfect matching in the bipartite graph G_B. Namely, for $i = 1, \ldots, 4$, the processor P_i holds the row blocks d_i and σ_i. Consider the column j such that $\pi_d(j) = d_3$, $\Lambda_d(j) = \{d_2, d_3\}$, and $\Lambda_\sigma(j) = \{\sigma_1, \sigma_3, \sigma_4\}$. The contribution of the column j to the cost of the matching M is two units: one for the edge (d_1, σ_1) and 1 for the edge (d_4, σ_4). Processor P_3 has to send x_j to processors P_1 (due to σ_1), P_2 (due to d_2), and P_4 (due to σ_4). The volume of send operation from P_3 to P_2 is captured by $C^{\lambda}(\Pi_d)$ since $d_2 \in \Lambda_d(j)$. The volumes of send operations from P_3 to P_1 and P_4 are captured by the weights of the matching edges (d_1, σ_1) and (d_4, σ_4), respectively.

4 Numerical experiments

The test matrices are shown in Table 1. They originate from semiconductor device simulation and are provided in the University of Florida sparse matrix collection by O. Schenk. The four matrices are rather representative of the whole matrix suite coming from the DESSIS simulator [17]. Their sizes, numbers of nonzeros, and problem dimensionality are shown in columns $\boldsymbol{n}$, $\boldsymbol{nnz}$, and `dim`, respectively. The column $\boldsymbol{\mathcal{D}}_{\mathtt{r;c}}$ shows the percentages of the $\boldsymbol{\mathcal{D}}$ rows and columns, respectively, as reported by the `info` routine from SPARSKIT [30]. The estimated condition numbers (column `condest`) have been reported first in [38], in which they have been computed using the MC71 algorithm [22] with the direct solver PARDISO [37]. Since these matrices are very ill-conditioned, they appear not manageable by off-the-shelf preconditioned iterative methods and direct solvers, as shown in [38] for a few sequential implementations, and may require preprocessing. In particular, the authors of [38] have employed a pre-ordering P_r of the unknowns of A to maximize the product of the diagonal values of P_rA followed by proper scaling. Consecutively, the condition number has been reduced and the diagonal dominance of rows and columns has increased. Note

Table 1: Matrix characteristics

`Matrix`	n	nnz	`dim`	`condest`	$\mathcal{D}_{\mathtt{r;c}}$, %
igbt3	10,938	234,006	3D	4.74e19	4 ; 26
2768_bjtcai	27,628	442,898	2D	6.46e19	49 ; 43
matrix_9	103,430	2,121,550	3D	3.08e23	35 ; 11
matrix_new_3	125,329	2,678,750	3D	3.48e22	63 ; 58

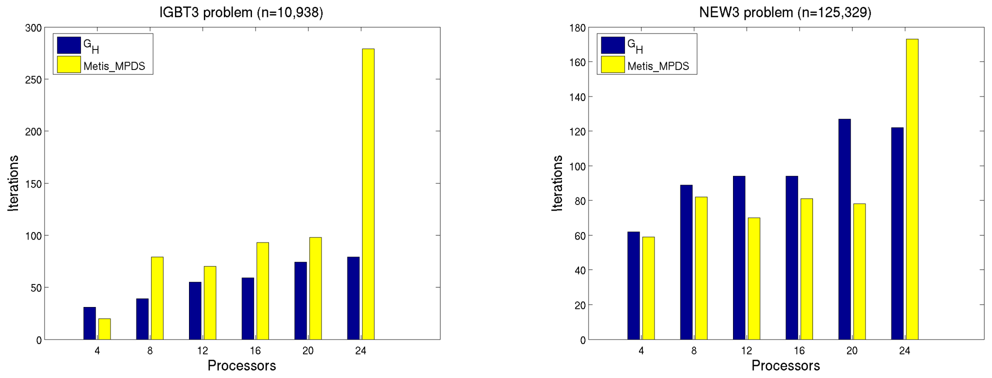


Figure 3: Symmetric partitioner and MC64 with scaling may lead to convergence

that this reordering is also implemented as MC64 algorithm in the HSL collection [22], with which we have also experimented. For parallel iterative methods, MC64 with scaling (MPDS) has lead to convergence when the symmetric partitioning MeTiS [23] has been used. Figure 3 shows the number of outer iterations for the MeTiS_MPDS combination as compared with the performance for the non-weighted hypergraph G_H. Note that no convergence has been observed with MeTiS otherwise. The results of applying MPDS to hypergraph partitioning were not as dramatic, however, and did not show radical improvement in the convergence process. To better study the effects of our partitioning schemes, we opted not to preprocess with MPDS, which may also be prohibitively expensive for large-scale matrices and less expensive procedures, such as ddPQ [32], could be used. The linear system tested have been scaled by 2-norms of rows followed by 2-norms of columns.

The experiments were performed on NERSC computer resources [1] using PaToH [9] hypergraph partitioner, which implements partitioning sequentially and thus has some limitations on the matrix size. To perform truly large-scale tests, a new Zoltan [12] hypergraph partitioning is already available [13] and will be used in our future work. We have used the default choices for most settings in PaToH partitioning except that the partitioning of superior quality `QUALITY` was asked and both cost objective functions $C^{\lambda}(\Pi)$ and $C^{c}(\Pi)$ (Equation (3)) were considered. For our two-constraint formulation (denoted as MC), the PaToH `MultiConst_Partition` function has been employed with default settings but except for the for coarsening algorithm, which ew set to the agglomerated version of the heavy connectivity matching (HCC) [9]. Note that in PaToH many parameter choices are based on randomized algorithms, e.g., the default for initial

[1]The NERSC IBM SP RS/6000, named Seaborg, is a distributed memory computer with 380 compute nodes with 16 processors per node. Each processor has a peak performance of 1.5 GFlops. The nodes having 16GBytes of shared memory were used.

partitioning parameter is a greedy algorithm starting with a randomly selected vertex. Thus, we have run all the experiments several times. Since no changes have been observed in the outer iteration numbers and the difference in the cutsize amounted to no more than 4%, the results presented in this Section are not averaged over all the runs.

To solve the distributed linear systems, the pARMS library [27] was employed. In particular, we have utilized its implementations of Additive Schwarz and distributed Schur complement preconditionings. Flexible GMRES [31] (FGMRES) was taken as iterative accelerator with the Krylov subspace of size twenty and two different convergence tolerances $t_1 = 10^{-7}$ and $t_2 = 10^{-4}$. The following list displays the preconditioner settings along with their shorthand names:

- `add_ilut` is non-overlapping Additive Schwarz procedure, with local incomplete LU with dual-threshold (ILUT) [31] on subdomains with fill-in parameter 60, dropping tolerance 10^{-3}, with and without five inner iterations,

- `lsch_ilut` is distributed Schur complement technique (dSC) [33] with ILUT as factorization used for local submatrices using the same settings as in `add_ilut`. This preconditioner is equivalent to the Additive Schwarz preconditioner on the Schur complement system.

4.1 Overall comparison of partitioning strategies

For two variations of the weight schemes (columns $W_h^{\hat{s}}$ and $W_h^{\bar{s}}$), the IPM algorithm (column `IPM`), the two-constraint partitioning (column `MC`), and the non-weighted standard hypergraph partitioning (column `nW`), Table 2 compares the outer iteration numbers to converge (columns I_O) and the preconditioner stability (columns $\mathcal{E}$) when `add_ilut` is used without overlap and having no inner iterations. The maximum number of iterations allowed was 700 with the convergence tolerance of $t_1 = 10^{-7}$. The partitioning threshold Δ was set to 0.8, and the results shown are the best among the two objective functions $C^{\lambda}(\Pi)$ and $C^{c}(\Pi)$ for each case. The smallest number of iterations is highlighted in bold face for each matrix/processor combination. Observe that `nW` has failed to converge for `matrix_new_3` on 12 processors and `MC` has performed poorly in most cases. The `MC` results were not obtained on 12 processors because `PaToH_MultiConst` partitions only into the power of two number of parts. There is also no capability to use weights with `PaToH_MultiConst`.

For large processor numbers, it becomes difficult for the simple Additive Schwarz preconditioning to produce convergence (see [39], e.g.) even with a sophisticated partitioning algorithm. A remedy might be to use a variant of distributed Schur Complement (dSC). Table 3 presents the results for the 32-processor experiment set-up similar to the one shown in Table 2 except that it uses the dSC, `lsch_ilut`, with five inner iterations as preconditioiner. For even larger processor numbers, larger matrices need to be considered, which we did not pursue in this initial study of the proposed algorithms. It may be observed in Table 3 that the proposed algorithms show an improvement in the outer iterations over the performance of `nW`. An exception is the convergence with IPM for the igbt3 problem.

4.2 Tuning weight assignment parameters

The implications for preconditioning caused by either the proposed hypergraph partitioning weight schemes or by the IPM algorithm depend on the hyperedge separation into $\boldsymbol{\mathcal{D}}$ and $\tilde{\boldsymbol{\mathcal{D}}}$ types, i.e., on the comparison of the relative diagonal dominance τ' with the threshold Δ for each hyperedge.

Table 2: Comparison of outer iterations for various partitioning strategies with $\Delta = 0.8$.

Matrix	nW		$W_h^{\hat{s}}$		$W_h^{\bar{s}}$		IPM		MC	
	I_O	$\mathcal{E}$	I_O	$\mathcal{E}$	I_O	$\mathcal{E}$	I_O	$\mathcal{E}$	I_O	$\mathcal{E}$
Four processors										
igbt3	**151**	4.7	157	4.7	153	4.7	176	4.7	700	4.69
2768_bjtcai	339	4.24	413	4.28	328	4.24	**211**	4.22	330	4.22
matrix_9	167	3.21	154	3.28	177	3.16	**150**	3.33	216	2.93
matrix_new_3	328	3.86	603	3.45	**211**	3.38	213	3.71	700	3.86
Eight processors										
igbt3	408	4.5	**322**	4.51	400	4.52	612	4.51	700	5.71
2768_bjtcai	466	4.24	655	4.12	359	4.23	**356**	4.23	700	4.22
matrix_9	227	3.13	180	3.25	237	3.16	**152**	3.21	300	2.86
matrix_new_3	379	3.41	675	3.33	**344**	3.84	373	3.87	700	3.92
Twelve processors										
igbt3	**196**	4.43	200	4.43	241	4.48	199	4.49		
2768_bjtcai	507	3.78	579	3.77	502	4.17	**450**	3.97		
matrix_9	216	3.21	**191**	3.23	207	3.16	218	2.99		
matrix_new_3	700	2.82	636	3.3	489	3.82	**479**	3.68		

Table 3: 32-processor experiment with distributed Schur Complement preconditioiner

Matrix	nW		$W_h^{\hat{s}}$		$W_h^{\bar{s}}$		IPM		MC	
	I_O	$\mathcal{E}$	I_O	$\mathcal{E}$	I_O	$\mathcal{E}$	I_O	$\mathcal{E}$	I_O	$\mathcal{E}$
igbt3	122	4.06	**73**	4.10	95	4.3	107	4.27	84	3.92
2768_bjtcai	140	3.74	**117**	3.75	133	3.75	**117**	4.08	180	3.49
matrix_9	40	3.29	37	3.26	42	3.39	**34**	3.44	74	3.53
matrix_new_3	113	3.47	110	3.59	96	3.61	**84**	3.85	144	3.50

Table 4: Varying the threshold Δ for assigning hyperedge weights. Iteration numbers to converge (left) and ratio of the $\mathcal{D}$ hyperedges to the total size of the corresponding hypergraph (right)

`Matrix`	Threshold $\boldsymbol{\Delta}$				W	C
	.2	**.5**	**.8**	**.9**		
Four processors						
igbt3	29	63	30	29	$\hat{s}$	λ
2768_bjtcai	58	35	60	60	$\hat{s}$	λ
matrix_9	46	66	47	52	$\hat{s}$	λ
matrix_new_3	97	75	58	58	$\bar{s}$	λ
Eight processors						
igbt3	37	77	37	34	$\hat{s}$	λ
2768_bjtcai	77	80	59	59	$\bar{\tau}$	λ
matrix_9	77	71	59	71	$\hat{\tau}$	λ
matrix_new_3	80	92	73	49	$\bar{\tau}$	c
Twelve processors						
igbt3	71	56	55	53	$\bar{s}$	λ
2768_bjtcai	79	80	64	64	$\bar{\tau}$	λ
matrix_9	89	75	75	75	$\hat{\tau}$	λ
matrix_new_3	110	96	82	67	$\bar{s}$	c

`Matrix`	Ratio of $\mathcal{D}$ hyperedges			
	.2	**.5**	**.8**	**.9**
igbt3	.86	.29	.0011	.0005
2768_bjtcai	.87	.35	.23	.23
matrix_9	.96	.17	.067	.05
matrix_new_3	.99	.54	.37	.37

Table 4(left) shows the iteration numbers for the `add_ilut` preconditioner with five inner iterations and FGMRES(20) with t_2 for $\Delta = \{0.2, 0.5, 0.8, 0.9\}$ (columns 2–5). The weight scheme is given in column W, where $\hat{s}$ and $\bar{s}$ refer to the W_h^s weight scheme with $\mathcal{D}$ and $\tilde{\mathcal{D}}$ type hyperedges, respectively, while $\hat{\tau}$ and $\bar{\tau}$ refer to the W_h^τ weight scheme with $\mathcal{D}$ and $\tilde{\mathcal{D}}$ type hyperedges, respectively. Column C shows the partitioning objective considered in each case, such that λ and c stand for $C^\lambda(\Pi)$and $C^c(\Pi)$, respectively.

Although the threshold Δ may be given as user-defined parameter and obviously depends on the matrix at hand, the partitioner may be outfitted with a certain pre-set value of Δ, which will be satisfactory for a large number of matrices. We have investigated several values and observed that Δ is sensitive only up to rather broad ranges. In particular, Table 4(right) implies, by proving the ratios of the number of $\mathcal{D}$ hyperedges to the total size of the hypergraph, that the constructed hypergraphs may be similar for $\Delta = 0.8$ and $\Delta = 0.9$ under the same weight schemes. Indeed the iteration numbers are similar in most cases for these thresholds (Table 4, left). The larger values of Δ — those closer to one — provide for a better convergence in most cases. These observations are similar in spirit to the findings in [32] concerning the non-symmetric reorderings to improve the preconditioning.

4.3 Hyperedge cutsizes

Both objective functions in (3) may include a weight term as multiplier cost c_j. Thus, PaToH returns the *weighted* partitioning cost by default. To find out the number of hyperedges cut, PaToH provides a separate function `Compute_Cut` that may be called on the already partitioned hypergraph with hyperedge weights unset. For 8- and 32-way partitionings, Figures 4 depicts — as groups of 10 bars — the unweighted hyperedge cutsizes normalized by the test problem size corresponding to each group. The darker-colored bars represent the cutsizes for the $C^\lambda(\Pi)$ objective function, whereas the lighter-colored bars are for $C^c(\Pi)$. It may be observed that, as expected,

Table 5: Different matchings for the bipartite graph G_B constructed in the IPM algorithm, when $\Delta = 0.5$ and $C^{\lambda}(\Pi)$ are used to partition separately hypergraphs G_{dd} and $G_{\sigma\sigma}$ into eight parts. Standard deviations are shown in columns `loc`

Matrix	`min_match`			`simple_match`		
	I_O	`loc`	$C_B(M)$	I_O	`loc`	$C_B(M)$
igbt3	266	.0017	5,054	306	.0019	9,417
2768_bjtcai	57	.0026	14,984	96	.0021	22,805
matrix_9	76	.0008	57,779	94	.0007	69,226
matrix_new_3	145	.0008	84,501	167	.0005	143,542

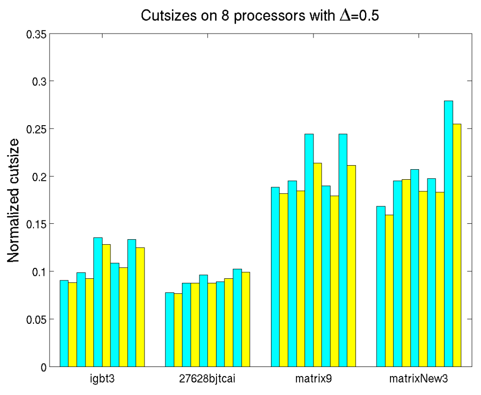
there are fewer cut hyperedges when the $C^c(\Pi)$ cost is used, and the difference is less pronounced for smaller problems. According to [8], however, the $C^{\lambda}(\Pi)$ (connectivity) objective function corresponds exactly to the total communication volume. In our experiments, $C^{\lambda}(\Pi)$ has lead to a better preconditioner performance more often as seen in Table 4 and, for the data in Table 3, $C^{\lambda}(\Pi)$ was chosen as best performing in 13 out of 16 entries. Within each bar group, a comparison of the hyperedge cuts shows that cutsize may be almost equal for the weight schemes (bars 2-9) and for standard (non-weighted) hypergraph representation (bars 0 and 1) when smaller problems are considered. The difference grows, however, with lowering the partitioning threshold Δ from 0.8 to 0.5 (Figures 4(a) and (b), respectively). The same performance is observed for the 32-way partitionings shown in Figures 4(c) and (d), respectively. For larger problem sizes, the weight schemes produce more hyperedge cuts in almost all the cases. In the bar groups `matrix9` and `matrixNew3`, the spiked cutsizes (bars 4 and 8) correspond to the $W_h^{\hat{s}}$ and $W_h^{\hat{\tau}}$ weight schemes, i.e., to the weight schemes with $\boldsymbol{\mathcal{D}}$ hyperedges being heavy. For the large partitioning threshold values, there are very few $\boldsymbol{\mathcal{D}}$ hyperedges, and thus the partitioner is free to cut a great deal of hyperedges under the $C^{\lambda}(\Pi)$ objective function, which is not the case for $C^c(\Pi)$ (bars 5 and 9). Consequently, to balance the "preconditioner" and "structural quality" viewpoints, it is better to use the $C^c(\Pi)$ objective function when the $W_h^{\hat{s}}$ and $W_h^{\hat{\tau}}$ weight schemes are applied. Lowering the partitioning threshold Δ yields more $\boldsymbol{\mathcal{D}}$ hyperedges and levels the cut discrepancies but not enough, especially for $W_h^{\hat{\tau}}$ in the `matrixNew3` group as seen in Figures 4(a) and (c), bars 8.

4.4 Detailed considerations for the IPM algorithm

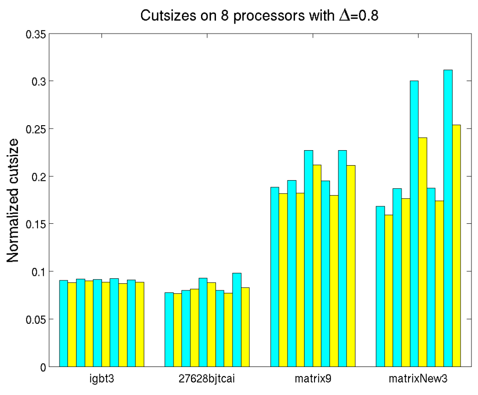
The IPM algorithm 3.1 finds minimum weight perfect matching on the bipartite graph G_B to reduce the communication volume among K parts. The ACM TOMS algorithm 548 [7] is considered in our experiments to obtain this matching.

Table 5 provides the results from using this algorithm on G_B of size $2K = 16$ with different definitions of the matching cost $C_B(M)$ (column `min_match`). Column `simple_match` gives the results for a simple matching of parts with the same indices, i.e., the parts d_i and $\sigma_i,\ i = 1, \ldots, K$ are matched. It may be observed that the `simple_match` is inferior to `min_match` in terms of the number of iterations (columns I_O) and the cost $C_B(M)$. Thus, the objective of reducing communication volume also benefits convergence by keeping together more strongly connected $\boldsymbol{\mathcal{D}}$ and $\tilde{\boldsymbol{\mathcal{D}}}$ parts. Columns `loc` show the standard deviations (STD) of the sizes of the local subdomains normalized by the average subdomain size for a given matrix. The average subdomain size appeared to be the same across all the cost metrics.

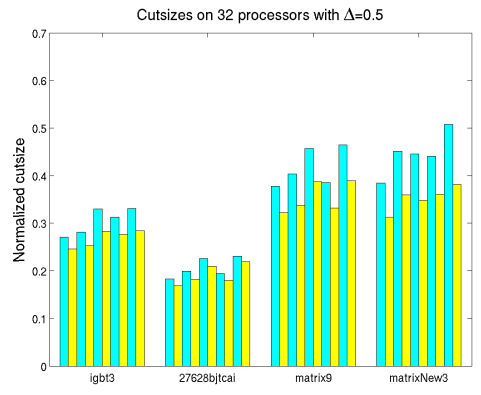
In Table 6, columns r$\boldsymbol{\mathcal{D}_r}$ (r$\boldsymbol{\mathcal{D}_c}$) show the averages over K ratios of $\boldsymbol{\mathcal{D}}$ rows (columns) to



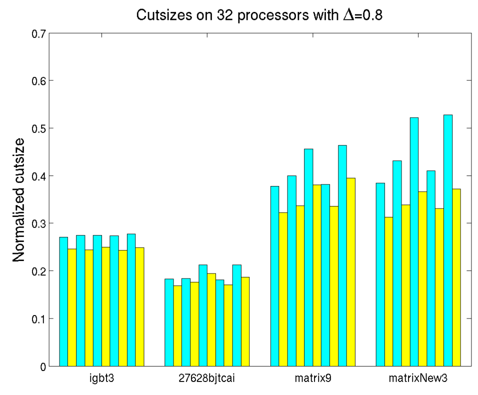
(a)



(b)



(c)



(d)

Figure 4: Hyperedge cutsizes for various weight schemes and two objective functions $C^{\lambda}(\Pi)$ and $C^{c}(\Pi)$. Darker-colored bars are for $C^{\lambda}(\Pi)$ and lighter-colored bars are for $C^{c}(\Pi)$. In each plot, four groups of bars represent the data for `igbt3`, `2768_bjtcai`, `matrix_9`, and `matrix_new_3`, respectively. In a group, the bars are ordered, from 0 to 9, such that each corresponds to a particular weight scheme as follows: Non-weighted (bars 0-1), $W_h^{\bar{s}}$ (bars 2-3), $W_h^{\hat{s}}$ (bars 4-5), $W_h^{\bar{\tau}}$ (bars 6-7), and $W_h^{\hat{\tau}}$ (bars 8-9). Figures (a) and (b) show the 8-way partitioner results; (c) and (d) show the 32-way ones, when different thresholds Δ are used

Table 6: Comparison of the balances of the $\boldsymbol{\mathcal{D}}$ rows and columns when $\Delta = 0.5$ and $C^c(\Pi)$ are used. Mean values followed by standard deviations are shown in Columns `IPM`, `nW`, and $W_h^{\bar{s}}$

`Matrix`	`IPM`		`nW`		$W_h^{\bar{s}}$	
	r$\boldsymbol{\mathcal{D}_r}$	r$\boldsymbol{\mathcal{D}_c}$	r$\boldsymbol{\mathcal{D}_r}$	r$\boldsymbol{\mathcal{D}_c}$,	r$\boldsymbol{\mathcal{D}_r}$	r$\boldsymbol{\mathcal{D}_c}$,
Four processors						
igbt3	.32; .0873	.36; .0451	.31; .1109	.28; .0954	.32; .1109	.28; .0968
2768_bjtcai	.33; .0377	.41; .0408	.33; .1431	.37; .1173	.33; .1812	.36; .1576
matrix_9	.29; .0206	.30; .0606	.29; .0141	.21; .1078	.29; .0299	.21; .1075
matrix_new_3	.56; .0173	.60; .0171	.55; .0206	.56; .0183	.55; .0126	.55; .0222
Eight processors						
igbt3	.05; .0198	.06; .0167	.32; .1296	.30; .1206	.32; .1184	.29; .0949
2768_bjtcai	.24; .0106	.52; .0119	.32; .2371	.38; .1971	.33; .2465	.37; .2079
matrix_9	.06; .0432	.16; .0238	.29; .0420	.25; .0968	.29; .0372	.24; .0922
matrix_new_3	.37; .0083	.41; .0083	.56; .0249	.58; .0158	.56; .0920	.57; .0838
Twelve processors						
igbt3	.32; .0807	.43; .0614	.32; .1204	.30; .1077	.32; .1240	.30; .1103
2768_bjtcai	.55; .0614	.64; .0308	.33; .2638	.38; .2289	.32; .2449	.37; .2125
matrix_9	.31; .0811	.43; .0509	.29; .0429	.25; .0929	.29; .0394	.25; .0918
matrix_new_3	.29; .0648	.64; .0235	.55; .0360	.58; .0308	.56; .1762	.58; .1612

the total local subdomain size followed by the their STD for the IPM algorithm and non-weighted hypergraph partitioning (columns `IPM` and `nW`, respectively).

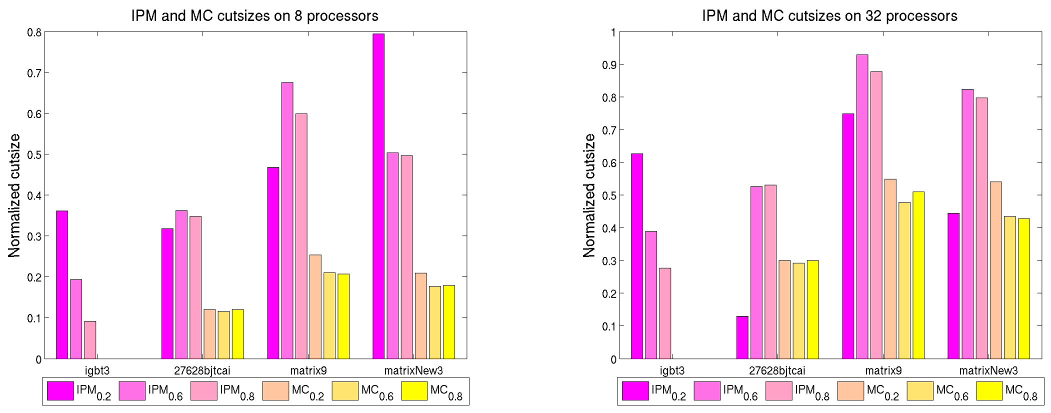
Observe that IPM produces generally more balanced numbers of both $\boldsymbol{\mathcal{D}}$ rows and columns, which is in agreement with its goal of balancing $\boldsymbol{\mathcal{D}}$ and $\tilde{\boldsymbol{\mathcal{D}}}$ columns/rows per processor. In addition, Table 6 shows r$\boldsymbol{\mathcal{D}_r}$ and r$\boldsymbol{\mathcal{D}_c}$ for the weight scheme $W_h^{\bar{s}}$, which appears to be representative of the other weight schemes tested and seems to be no better balanced than `nW`.

Section 4.2 presented some evidence (Table 4) that larger values of Δ in weight schemes may be more beneficial for iterative convergence. The same argument holds for the IPM algorithm since it also distinguishes $\boldsymbol{\mathcal{D}}$ and $\tilde{\boldsymbol{\mathcal{D}}}$ hyperedges as the fist step. Table 7 shows the outer iteration (columns I_0) numbers with the same iterative solver parameters as in Section 4.2 and with the IPM algorithm used for partitioning. It is also interesting to note how the relative cutsizes of the partitions Π_{dd} and $\Pi_{\sigma\sigma}$ change as Δ increases. As seen in columns I_O, this is actually beneficial for the iterative convergence even though the "diagonal dominant" hypergraph G_{dd} becomes small as Δ increases.

For the IPM algorithm, Figure 5 shows total communication volumes $C_{IPM} = C^{\lambda}(\Pi_d) + C^{\lambda}(\Pi_\sigma) + C_B(M)$, as given by Theorem 3.1, and C_{MC} normalized by the problem sizes. C_{MC} is generally better except on the `igbt3` problem, for which it has produced very unbalanced partitions. (For the sake of plot scale, C_{MC} are not presented for `igbt3`.) However, for 32-way partitioning IPM results in smaller cutsizes on some threshold Δ values. Note that, for larger processor numbers (Figure 5(b)), larger values of Δ yield lesser communication volume.

Table 7: Outer iteration numbers and ratios of cutsizes, $C^\lambda(\Pi_{dd})/C^\lambda(\Pi_{\sigma\sigma})$ on 8 processors when the IPM partitioning algorithm is used with the threshold $\Delta \in \{.2, .6, .8, .9\}$

`Matrix`	**.2**		**.6**		**.8**		**.9**	
	I_O	rC	I_O	rC	I_O	rC	I_O	rC
igbt3	700	6.96	38	.26	35	.02	36	.002
2768_bjtcai	700	7.43	56	.48	63	.51	63	.51
matrix_9	495	94.89	68	.06	66	.07	69	.08
matrix_new_3	700	255.65	66	.74	83	.71	68	.72



(a) (b)

Figure 5: Separate partitioning techniques: Comparison of cutsize measures for $\Delta \in \{.2, .6, .8\}$ and $C^\lambda(\Pi)$

5 Conclusions

We have shown that partitioning can affect convergence properties of distributed iterative solution algorithms. In particular, for difficult linear systems, preconditioning may be more stable when subdomains assigned to a processor are more diagonally-dominant and nodes on the interfaces are rather small. We have shown several strategies for incorporating numerical values of the coefficient matrix into hypergraph models by means of weights on hyperedges using the concept of weak diagonal-dominance ($\mathcal{D}$). Several weight schemes constructed show good performance on difficult circuit device simulation matrices.

The goal of achieving a good convergence may be viewed as a complex objective function for a partitioning algorithm and may be attained in stages: first partition to balance weak diagonal dominance among subparts then merge the subparts. We have presented one such an algorithm (called IPM) and showed that it balances the parts well according to the $\mathcal{D}$ property. In addition, we have proved that it correctly minimizes the total communication volume when a minimum weight matching problem is defined in a specific way on the bipartite graph of $\mathcal{D}$ and $\tilde{\mathcal{D}}$ parts. The proposed algorithms have been studied with respect to the threshold parameter used to classify the hyperedges as $\mathcal{D}$ and $\tilde{\mathcal{D}}$ and we found that larger values of this parameter are typically more beneficial for convergence.

Our future work includes conducting large-scale experiments and using the proposed algorithms with parallel hypergraph partitioning codes, such as Zoltan.

References

[1] C. Aykanat, B. B. Cambazoglu, and B. Uçar. Multilevel direct k-way hypergraph partitioning with multiple constraints and fixed vertices. *J. Parallel and Distr. Com.*, submitted, 2006.

[2] C. Aykanat, A. Pinar, and Ü. V. Çatalyürek. Permuting sparse rectangular matrices into block-diagonal form. *SIAM J. Sci. Comput.*, 25(6):1860–1879, 2004.

[3] R. H. Bisseling and W. Meesen. Communication balancing in parallel sparse matrix-vector multiplication. *Electronic Transactions on Numerical Analysis*, 21:47–65, 2005.

[4] M. Bollhöfer. A robust ILU with pivoting based on monitoring the growth of the inverse factors. *Linear Algebra and its Applications*, 338(1-3):201–213, 2001.

[5] M. Bollhöfer and Y. Saad. Multilevel preconditioners constructed from inverse–based ILUs. *SIAM Journal on Scientific Computing*, 27:1627–1650, 2006.

[6] X. C. Cai and M. Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM Journal on Scientific Computing*, 21:792–797, 1999.

[7] Giorgio Carpaneto and Paolo Toth. Algorithm 548: Solution of the assignment problem [H]. *ACM Transactions on Mathematical Software*, 6(1):104–111, March 1980.

[8] Ü. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. Parallel Distrib. Syst.*, 10(7):673–693, 1999.

[9] Ü. V. Çatalyürek and C. Aykanat. PaToH: A multilevel hypergraph partitioning tool, version 3.0. Technical report, Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey, 1999.

[10] Ü. V. Çatalyürek and C. Aykanat. A hypergraph-partitioning approach for coarse-grain decomposition. In *Proceedings of Scientific Computing 2001 (SC2001),* pages 10–16, Denver, Colorado, November 2001.

[11] E. Chow and Y. Saad. Experimental study of ILU preconditioners for indefinite matrices. *Journal of Computational and Applied Mathematics*, 86:387–414, 1997.

[12] K.D. Devine, E.G. Boman, R.T. Heapby, B. Hendrickson, and C. Vaughan. Zoltan data management service for parallel dynamic applications. *Computing in Science and Engg.*, 4(2):90–97, 2002.

[13] K.D. Devine, E.G. Boman, R.T. Heaphy, R.H. Bisseling, and U.V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE, 2006.

[14] I. S. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 20:889–901, 1999.

[15] I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22(4):973–996, 2001.

[16] I. S. Duff, S. Riyavong, and M. B. van Gijzen. Parallel preconditioners based on partitioning sparse matrices. Technical Report TR/PA/04/114, CERFACS, Toulouse, France, 2004.

[17] Integrated Systems Engineering. DESSIS_ISE: Reference manual. http://www.ise.com, 2003.

[18] B. Hendrickson. Graph partitioning and parallel solvers: has the emperor no clothes? *Lect. Notes Comput. Sci.*, 1457:218–225, 1998.

[19] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26(12):1519–1534, 2000.

[20] B. Hendrickson and T. G. Kolda. Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing. *SIAM J. Sci. Comput.*, 21(6):2048–2072, 2000.

[21] B. Hendrickson and R. Leland. The Chaco user's guide — version, 1994.

[22] HSL:a collection of iso fortran codes for large scale scientific computation. http://www.cse.clrc.ac.uk/nag/hsl/hsl.shtml.

[23] G. Karypis and V. Kumar. MeTiS, unstructured graph partitioning and sparse matrix ordering system. version 2.0. Technical report, University of Minnesota, Department of Computer Science, Minneapolis, MN 55455, August 1995.

[24] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. Technical Report 98-019, University of Minnesota, Department of Computer Science/Army HPC Research Center, May 1998.

[25] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint hypergraph partitioning. Technical Report 99-034, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN 55455, November 1998.

[26] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. Wiley–Teubner, Chichester, U.K., 1990.

[27] Z. Li, Y. Saad, and M. Sosonkina. pARMS: A parallel version of the algebraic recursive multilevel solver. *Numerical Linear Algebra with Applications*, 10:485–509, 2003.

[28] M. Olschowska and A. Neumaier. A new pivoting strategy for Gaussian elimination. *Linear Algebra and its Applications*, 240:131–151, 1996.

[29] A. Pinar and B. Hendrickson. Partitioning for complex objectives. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2001.

[30] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical report, Technical Report RIACS-90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA, 1990.

[31] Y. Saad. *Iterative Methods for Sparse Linear Systems, 2nd edition*. SIAM, Philadelpha, PA, 2003.

[32] Y. Saad. Multilevel ILU with reorderings for diagonal dominance. *SIAM Journal on Scientific Computing*, 27(3):1032–1057, 2005.

[33] Y. Saad and M. Sosonkina. Distributed Schur Complement techniques for general sparse linear systems. *SIAM J. Scientific Computing*, 21(4):1337–1356, 1999.

[34] Y. Saad and M. Sosonkina. Non-standard parallel solution strategies for distributed sparse linear systems. In A. Uhl P. Zinterhof, M. Vajtersic, editor, *Parallel Computation: Proc. of ACPC'99*, Lecture Notes in Computer Science, Berlin, 1999. Springer-Verlag.

[35] Y. Saad, M. Sosonkina, and J. Zhang. Domain decomposition and multi-level type techniques for general sparse linear systems. In *Domain Decomposition Methods 10*, Providence, RI, 1998. American Mathematical Society.

[36] Y. Saad and B. Suchomel. ARMS: An algebraic recursive multilevel solver for general sparse linear systems. *Numerical Linear Algebra with Applications*, 9, 2002.

[37] O. Schenk, K. Gärtner, W. Fichtner, and A. Stricker. PARDISO: a high-performance serial and parallel sparse linear solver in semiconductor device simulation. *Future Generation Computer Systems*, 18(1):69–78, 2001.

[38] O. Schenk, S. Röllin, and A. Gupta. The effects of unsymmetric matrix permutations and scalings in semiconductor device and circuit simulation. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23(3):400–411, 2004.

[39] B. Smith, P. Bjørstad, and W. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, New York, 1996.

[40] M. Sosonkina, Y. Saad, and X. Cai. Using the parallel algebraic recursive multilevel solver in modern physical applications. *Future Generation Computer Systems*, 20:489–500, 2004.

[41] B. Uçar and C. Aykanat. Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. *SIAM J. Sci. Comput.*, 25(6):1827–1859, 2004.

[42] B. Uçar and C. Aykanat. Revisiting hypergraph models for sparse matrix partitioning. *SIAM Rev.*, accepted for publication, 2006.

[43] B. Uçar and C. Aykanat. Partitioning sparse matrices for parallel preconditioned iterative methods. *SIAM J. Sci. Comput.*, submitted, 2004.

[44] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.

[45] C. Walshaw, M. Cross, and K. McManus. Multiphase mesh partitioning. *Appl. Math. Model.*, 25:123–140, 2000.